FERMIUM: A Framework for Real-time Procedural Point Cloud Animation and Morphing

Ole Wegen[ID], Florence Böttger, Jürgen Döllner and Matthias Trapp[ID]

Hasso Plattner Institute, Faculty of Digital Engineering, Unversity of Potsdam, Germany

Figure 1: *Single frames from a octree-based point cloud morphing animation created using the FERMIUM framework.*

Abstract

This paper presents a framework for generating real-time procedural animations and morphing of 3D point clouds. Point clouds or point-based geometry of varying density can easily be acquired using LiDAR cameras or modern smartphones with LiDAR sensors. This raises the question how this raw data can directly be used in the creative industry to create novel digital content using animations. For this purpose, we describe a framework that enables the implementation and combination of animation effects for point clouds. It takes advantage of graphics hardware capabilities and enables the processing of complex datasets comprising up to millions of points. In addition, we compare and evaluate implementation variants for the subsequent morphing of multiple 3D point clouds.

CCS Concepts

• *Computing methodologies* ,..., *Procedural animation; Point-based models; Graphics processors;*

1. Introduction

Point-based geometry represented as 3D Point Clouds (PCs) are sets comprising a large number of attributed 3D points. Similar to 3D polygonal meshes, their attributes can comprise color, surface normals, reflectiveness coefficients, segment identifiers, as well as time. PCs are a simple, compact, and flexible geometric representation that can easily be acquired for real-world scenes using off-the-shelf photogrammetric techniques. In addition, high-end consumer smartphones with Light Detection And Ranging (LiDAR) sensors (e.g., iPad Pro or iPhone 12) enable a straightforward acquisition. Further, PCs can be obtained by sampling 3D polygonal geometry.

Surprisingly, and despite its potential applicability in various domains, their usage as a basic geometric representation in computer animation and respective animation systems or frameworks are sparsely studied. Such systems can be applied in previsualization (previz) or Digital Content Creation (DCC) to evaluate ideas and

concepts early before performing possibly costly data enhancement or transformation (e.g., synthesizing textured 3D meshes). In computer games, for example, 3D PCs enable effects and animations, such as particle systems or morphing, to scanned assets [SPB*19]. With respect to this, we present implementation approaches for procedural animations [Ebe14] of PCs.

Challenges for Rendering 3D Point Cloud Animations. Regarding animation, especially morphing (Fig. 1), PCs have an advantage over 3D polygonal models: they do not comprise inherent connectivity information that needs to be respected or maintained, thus in turn facilitate their representation, transformation, and rendering. Given a sufficient point density, PCs can also yield high quality renderings [SW15]. In contrast thereto, however, PCs are often characterized by their complexity, i.e., the number of points and the number of per-point attributes. While the rendering of static 3D PCs can be implemented straight-forward using Graphics Process-

ing Unit (GPU)-aligned rendering pipelines, the implementation of interactive animation techniques are challenging with respect to:

Efficient Data Handling (C1): An efficient representation of complex PCs and respective animation data allows for independent animation of their attributes. This also concerns the reduction of data transfer and update latencies to support real-time rendering for interactive control.

Interfacing Animation and Rendering (C2): Decoupling of processing operations that are required for animation and rendering (e.g., point-splat functions [ARLP18]) facilitates the interchange of respective techniques and increases ease-of-use.

Combination of Animation Techniques (C3): A potential combination and interchange of techniques within a common framework facilitates prototyping and development of new techniques and can reduce time to market.

Existing approaches to implement animation techniques for PCs are (1) customized integrations into existing game engines (to yield real-time rendering) or (2) specific tooling or scripts to extend existing software (e.g., Blender). These, however, require either auxiliary constructs for data representation (e.g., PCs represented as textures), lack real-time rendering capabilities, or can hardly handle PCs with millions of points.

Approach & Contributions. With respect to the challenges above, we present a unified GPU-aligned framework that enables real-time animation and morphing of complex PCs. The framework encapsulates data buffers and operating stages for both, animation and rendering of PCs that potentially comprise millions of points. By implementing animation and rendering techniques using Compute Shader (CS) programs operating on (atomically) writable and readable data buffers (Shader Storage Buffer Object (SSBO)), we achieve decoupling between multiple, simultaneously active animation techniques (C3) and rendering or stylization techniques (C2). Using individual data streams for PC attributes achieves a compact representation in Video RAM (VRAM) and allows for fine-granular data updates (C1). The framework components enable forward and deferred rendering techniques to be combined with point-attribute animations and the morphing of PCs with different sizes. To summarize, this paper makes the following contributions to the reader: (1) it presents a unified GPU-aligned framework that enables the implementation of various PC animation techniques in real-time and (2) it demonstrates its capabilities using different applications, such as per-point attribute animations and PC morphing.

2. Related Work

Animation Techniques for Point Clouds. Most previous work for real-time PC animation is concerned with physics-based simulation rather than the utilization of PCs for animation purposes. In the domain of physics-based simulation, point-based representations are already used for many years, for example in Harlow's work on fluid dynamics in 1962 [Har62]. On one hand, particle systems are used for simulating and depicting of natural phenomena such as fire, smoke, or fluids, as highlighted by the survey of Xi *et al.* [XLF*19] on the advances regarding smooth particle hydrodynamics and particle systems. On the other hand, apart from

particle systems, PCs are also used to represent deformable geometry in physic-based simulations. These meshfree or meshless approaches often use sparse PCs for representing the volume of an object. These volume points (also denoted as phyxels or particles) are used as simulation nodes. To convey the impression of surface smoothness, a dense PC consisting of surfels [PZvBG00], can be used for representing an object's surface. Bart Adams [Ada06] presents an overview of possible point-based rendering and animation approaches and demonstrates physics-based simulation of elastic or fracturing objects and fluids. Müller *et al.* also apply physically-based deformations to such point-based objects. Their system is capable of animating elastic, plastic, solidifying and melting objects based on the simulation of continuum mechanics. For real-time interaction, low-resolution models are used, while higher resolutions are possible for offline rendering [MKN*04]. Dharma *et al.* implement a fluid simulation through particle animation based on compute shaders [DJKM17]. They are able to animate 1 million particles at 50 Frames-per-Second (FPS) on a NVIDIA GTX Titan X GPU. In this work, we also rely on compute shaders for animating PCs. As we are concerned with PC animation for aesthetic purposes, which is not as complex as for example fluid simulation, we are able to render even larger PCs.

Morphing Techniques for Point Clouds. The goal of PC morphing is to transform the points of one PC over time in such a way that they form another, predefined PC. An important aspect of PC morphing is to find an appropriate mapping from a source PC to a destination PC. As one of the earliest publications on this topic, Čmolìk and Uller analyzed and compared different clustering methods based on Binary Space Partition (BSP) trees [CU03]. While this is a straight-forward approach, it does not allow feature-preserving morphing. Tian *et al.* compute a point mapping by forming a super PC by aligning the source and destination PCs in space. Subsequent clustering and local mapping then yields the final mapping result [THCF06]. Other publications approach the problem of finding a feature-preserving point mapping supported by user interaction by (1) letting a user define a mapping between selected features and (2) computing the mapping between single points. Xiao *et al.* compute the point mapping by projecting the PCs onto unit disks and aligning these according to corresponding features selected by the user [XZPF04]. A Level-of-Detail (LOD) technique is used to accelerate the parametrization, i.e., complex geometry is decomposed into several patches, that are treated independently. Aiming to achieve visually smooth morphing sequences, a similar approach is used by Wang *et al.* who use a unit sphere for parametrization instead of a unit disk. For the actual interpolation of the shapes, Laplacian coordinates are used [WZH12].

While most of the research focus on finding a meaningful mapping between the points, others focus on computing physically meaningful transition paths between the points. Bao *et al.* handle this problem as a physics-based energy optimization, using surface deformation analysis on a subset of the points [BGQ05]. Tan *et al.* approach the problem of finding a convincing morphing path by means of interpolation of vertex deformation gradients [TZZ09]. Chen *et al.* [CHG*20] presented an application example for PC morphing, by utilizing it for training data augmentation in the area of machine learning. A shortest-path computation
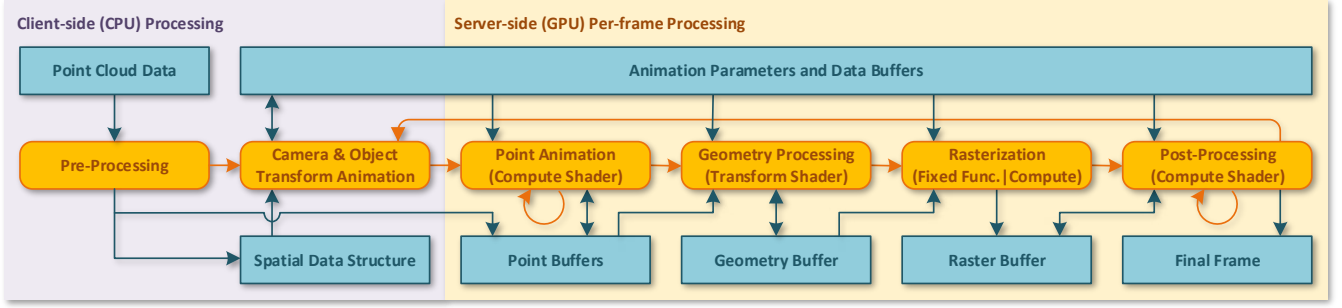
Figure 2: *Schematic overview of the compute components with control flow (orange) and data with data flow (blue) of our GPU-aligned framework for point cloud animation.*

is used to preserve the shapes of source and target PC as close as possible. The results are a number of static, intermediate PCs that can be used for training deep neural networks. Another application in the domain of virtual surgery simulation is presented by Cheng *et al.* [CSY*20]. They use a morphing approach to simulate deformation of soft tissue, such as organs. Their work therefore is more related to physics-based simulation, as morphing takes place between two configurations of the same PC.

3. Point Cloud Animation Framework

3D Point Cloud Structure & Data Acquisition. Throughout this paper, we make the following assumptions with respect to the representation, structure, and storage of a PC. We consider a PC $S = \{p_0, \ldots, p_N\}$ as discrete subset of the space $\mathbb{R}^{d_0} \times \ldots \times \mathbb{R}^{d_n}$ with the number of elements $|S| = N$. Each point $p \in S$ is given as a n-tuple of attributes $p = (a_0, \ldots, a_n)$, whose components are in the respective space:

$$p_i = \left(a_i^{(0)}, \ldots, a_i^{(n)} \right) \in \mathbb{R}^{d_0} \times \ldots \times \mathbb{R}^{d_n} \quad \text{for } 0 \leq i \leq N \quad (1)$$

$$a_i^{(j)} = \left(a_i^{(j,0)}, \ldots, a_i^{(j,d_j)} \right) \in \mathbb{R}^{d_j} \quad \text{for } 0 \leq j \leq n \quad (2)$$

The attribute $a_0 = (x, y, z) \in \mathbb{R}^3$ defines the point position in a global 3D coordinate reference system. In combination with additional attributes, such as colors or normal vectors, these attributes constitute a PC *configuration C*. For applications involving multiple PCs (e.g., PC morphing), we assume that these have the same configuration. With respect to memory layout, both for Central Processing Unit (CPU) and GPU, we choose Structure-of-Arrays (SoA) over Array-of-Structures (AoS) for attribute encoding. This facilitates the manipulation and exchange of individual attribute streams $a_0^{(i)}, \ldots, a_N^{(i)}$ in a configuration while preserving the states of others and allows for memory management on a per-attribute stream level.

The PC data used in this paper is acquired using two methods: (1) using the built-in LiDAR scanner of modern iOS devices and (2) by sampling 3D meshes. For sampling 3D meshes, our framework supports the following approaches and its combinations:

Vertex-based Sampling (VBS): The resulting PC is constructed by interpreting each vertex of the input mesh as a single point. The density of the PC depends on the number and distribution of input vertices.

Primitive-based Sampling (PBS): For each primitive, one point is added to the PC (e.g., the centroid of the triangle). The density of the PC then depends on the number and size of the triangles.

Density-based Sampling (DBS): Each triangle is sampled uniformly [OFCD02], according to a defined density value d (corresponding to the points per unit area), i.e., that larger triangles result in more points, than smaller triangles.

While VBS and PBS result in a fixed number of points for a specific mesh, DBS can be used to obtain PCs of different sizes. To achieve a smooth surface appearance, we use DBS with d between 200 and 300 for most of the examples in this paper (with the exception of Fig. 3(c), which uses $d = 100\,000$).

Overview of Framework Components. Fig. 2 shows the conceptual unified and GPU-aligned rendering pipeline that enables the combination of attribute animations (Sec. 4) and morphing (Sec. 5). We implemented the framework using C++ and OpenGL. It comprises the following main processing stages:

Pre-Processing: After loading a PC, this stage computes required scene statistics (e.g., bounding volumes) for camera and object animations as well as spatial data structures for point mapping when using a morphing animation. Simultaneously, the attribute point buffers are transferred to VRAM.

Camera & Object Transform Animation: This stage controls the animations of the virtual camera and the respective transformations of the PCs (e.g., translation, rotation, or scaling) using key frames. The animation data is encoded using uniform buffers and can be accessed in the subsequent point animation and geometry processing stage.

Point Animation: Passes in this stage implement attribute animations (Sec. 4) and morphing (Sec. 5) using possibly multiple compute shader invocations to modify the point buffers. This way, both types of animation techniques can be combined (C3).

Geometry Processing: Prior to rasterization, this stage implements primitive conversions (e.g., point to splats), geometry transformation and projection, as well as attribute mappings for texturing and shading (C2).
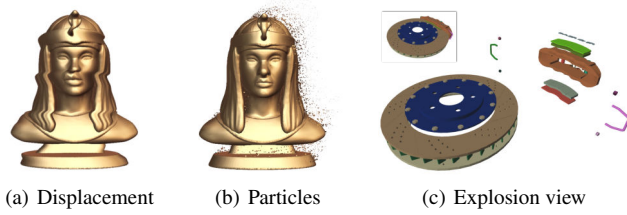
(a) Displacement   (b) Particles   (c) Explosion view

Figure 3: *Single frames from exemplary attribute animations implemented using our framework. The stateless (a) and stateful (b) attribute animation use a point cloud comprising* 673 411 *points while the explosion view animates* 1 577 299 *points.*

Rasterization: The rasterization stage performs an offscreen-rendering pass to create the raster buffers. It can be implemented as render-to-texture pass using fixed-function rasterization for rendering into G-Buffers or using compute shaders in combination with SSBOs [SKW21] (C2).

Post-Processing: This optional stage allows the integration of raster-based procedural animation effects that can be implemented by potentially using multiple passes.

Most animation parameters and data that are required by the individual stages are represented using suitable GPU-based data structures residing in VRAM. This comprises precomputed volumetric noise textures, raster-based representation of easing functions, or material captures for shading and texturing.

4. Attribute Animations

Attribute animations concerns the procedural modification of PC attributes such as position or color. Depending on the type of attribute animation, our framework implements these based on a combination of point buffers and compute shader functionality. Our framework supports the following attribute-animation types:

Stateless Attribute-Animation: The visual results of this type are solely determined by the input time and do not change point attributes persistently (Fig. 3(a)). It can be implemented by means of time-controlled shader programs and is suitable for simple animations that do not rely on prior animation results or simulation computations.

Stateful Attribute-Animation: This type relies on the time and prior animation steps. During animation, the attribute values are changed persistently (Fig. 3(b)). Stateful animations are especially useful for simulation of physics-based phenomena that are computed iteratively.

By invoking successive computer shader implementations in the point animation stage, instances of both animation types can be easily combined (C3, please refer to accompanying video).

Our framework supports the application of different easing functions per attribute class. This enables fine control over the animation behavior and allows decoupling time control between attribute classes. Each attribute class a_i can be interpolated using individual mapping functions. For any $1 \leq j \leq n$, we have a family of maps parametrized by time $t \in \mathbb{R}$ with $0 \leq t \leq 1$: $f_t^j : \mathbb{R}^{d_j} \to \mathbb{R}^{d_j}$. A

GPU-aligned animation framework can require the application of easing functions on client-side (CPU) and on server-side (GPU). The framework supports the definition of easing functions represented as Bézier curves [IKS20]. To enable efficient usage in the programmable shader stages, we encode these functions using a texture atlas that can be sampled and filtered (C1).

Further, our framework enables the combination of stateless attribute-animations with additional data on a per-vertex or per-group level. Assuming a segmented PC, these segment points can be animated in such a way that the segments move away from the center of the PC, potentially depicting it's internal structure (Fig. 3(c)). Such explosion views can be easily implemented using our framework by performing a compute shader pass for stateless attribute-animation. For example, the center of each segment and the center of the complete PC is computed. Subsequently, the translation vector for each segment can be computed and used for animation (please refer to accompanying video). This technique can be applied to implement animated transitions for PC visualization techniques in geo- or scientific visualization.

5. Morphing Animations

Morphing as an animation that changes (or morphs) one shape into another via a seamless transition, requires the processing of multiple PCs. In general, the morphing between two PCs is the animation during which the point attributes of the source S are interpolated to the points of a destination D. The morphing methods should consider the attributes configuration of the PCs and should be able to perform morphing between two different-sized PCs. The fundamental principle underlying all introduced methods is the finding of a mapping [CU03]. A PC morphing is defined component-wise by the mapping $F_t : S \to D$ with $t \in \mathbb{R}, 0 \leq t \leq 1$:

$$F_t(p_i) = q_i = F_t\left(a_i^{(1)}, \ldots, a_i^{(n)}\right) = \left(f_t^1\left(a_i^{(1)}\right), \ldots, f_t^n\left(a_i^{(n)}\right)\right)$$

A morphing sequence M is the successive morphing of a PC sequence $L = S_0, \ldots, S_k$ denoted by:

$$M = F_t^{(0)}, \ldots, F_t^{(k-1)} \quad \text{with} \quad F_t^{(i)} : S_i \to S_{i+1}$$

For testing and demonstration purposes, we implemented different approaches for computing a mapping F_t:

Random Mapping: The points of the source and destination PCs are shuffled and mapped to each other according to this order.

Axis-based Mapping: The points of both input PCs are sorted with respect to a reference axis. Subsequently, the points are mapped to each other using this new order.

Octree-based Mapping: For each of the PCs, an octree is constructed. Subsequently, the points are mapped according to a depth-first traversal of the leaf nodes of the octrees.

Distance-based Mapping: For each point of the source PC, the nearest-neighbor point of the destination PC is assigned. For nearest-neighbor search, a k-d tree can be used.

For these mappings, we have to consider the common case of different-sized PCs. For the first three approaches an assignment rate: $r = {\min(|S|,|D|)}/{\max(|S|,|D|)}$ is computed first. We then assign each point of the larger PC at index i the point of the smaller PC

at index $\lfloor i \cdot r \rfloor$. For the distance-based mapping, we first assign the points of the smaller PC to spatially close points of the larger PC. The not yet assigned points of the larger PC are then assigned to close points of the smaller PC. This way it is guaranteed that each point of the smaller PC is mapped to at least one point in the larger PC. After a mapping has been computed, the morphing can be performed. During morphing, the points of the larger PC are rendered and interpolated using an arbitrary easing function. The interpolation direction depends on the order of the PCs.

6. Variants for GPU-based Morphing Implementation

We implemented three different approaches to perform the morphing. Algo. 1 shows their common structure. The function calls shown in blue mark hook-methods that differ in their concrete implementation across the three approaches. The following functions are used in the code and not further defined here with respect to concrete implementation: ELAPSEDTIME returns the elapsed time since the last call; SHOULDSTARTMORPHING returns true if the morphing sequence should be started. SHOULDCHANGEMAPPING returns true if the mapping should be re-computed, which is the case if the mapping type is changed. The algorithm frame-

---

Algorithm 1 Common structure for morphing implementations.

```
 1: procedure MORPHSTEP(S, D, i)
 2:     INITMORPHSTEP(S, D)
 3:     BINDBUFFERS(i)
 4:     t ← 0
 5:     while t <= 1.0 do                    ▷ we assume duration of 1
 6:         PERFORMMORPHING(t)
 7:         t ← t + ELAPSEDTIME()
 8:     end while
 9: end procedure
10: procedure MORPH(L)
11:     INITPCS(L)
12:     COMPUTEMAPPING(L)
13:     while True do
14:         if SHOULDSTARTMORPHING() then
15:             for i ∈ [0, |L| − 2] do
16:                 MORPHSTEP(L[i], L[i + 1], i)
17:             end for
18:         end if
19:         if SHOULDCHANGEMAPPING() then
20:             COMPUTEMAPPING(L)
21:         end if
22:     end while
23: end procedure
```

---

work uses the following functions for buffer management. TRANSFERTOGPU transfers data of one or more buffers from Random-Access Memory (RAM) to VRAM. The CONSTRUCTBUFFER function creates a client-side buffer for a PC. The CONSTRUCTSORTEDBUFFER does the same but sorts the points according to the current morphing mapping type beforehand (this depends on the largest PC and is only used for the Vertex Buffer Object (VBO)-based implementation. The function INITBUFFEROFSIZE(x) initializes an empty client-side buffer of size x. Further, BINDBUFFERS(i) binds the GPU-side attribute buffers for the current

morphing step and PADBUFFER(B) adds the necessary padding to a buffer B that is to be used as a SSBO. The MAPPCS function returns a list of destination point indices to which the points of a source PC map ($\widehat{=} F_t$), depending on the mapping type. Finally, RENDERPASS issues a GPU-based rendering pass that uses the rasterization pipeline and the COMPUTEPASS function issues a compute shader pass without rasterizing any primitives.

VBO-based Implementation. For this approach, the source and destination point positions are written into buffers (according to the computed mapping), transferred to VRAM, and bound as VBOs. A vertex shader performs the interpolation between source and destination according to the current interpolation progress t. This approach is straightforward, but a re-transfer of the source and destination buffers are required, each time the morphing mapping changes. Algo. 2 shows the pseudocode implementation for the corresponding hook methods.

Algorithm 2 VBO-based morphing implementation.

```
 1: procedure COMPUTEMAPPING(L)
 2:     Buffer_S ← CONSTRUCTBUFFER(L[0])
 3:     TRANSFERTOGPU(Buffer_S)
 4:     for all S ∈ L_{\L[0]} do
 5:         Buffer_S ← CONSTRUCTSORTEDBUFFER(L[0], S)
 6:         TRANSFERTOGPU(Buffer_S)
 7:     end for
 8: end procedure
 9: procedure PERFORMMORPHING(t)
10:     RENDERPASS(t)
11: end procedure
```

SSBO-based Implementation. For the second approach, the points of the source and destination PCs are transferred only once to VRAM. Each time the mapping is computed, only this data is transferred to VRAM as a vector of buffer indices (C1). During rendering in the geometry processing stage, the vertex shader then has two additional inputs: (1) a SSBO containing the destination points and (2) the index into this buffer for fetching the current vertex. This trades the storage of additional data and higher initialization costs, for reduced update latency. Algo. 3 shows the pseudocode implementation for the corresponding hook methods.

CS-based Implementation. For the third approach, the interpolation between source and destination points is implemented using a separate compute shader pass. Similar to the SSBO-based approach, the PCs are transferred to the VRAM only once during preprocessing. Similarly, only the respective mapping data has to transmitted. The compute shader is invoked once for each vertex and stores the interpolated position in a result buffer that is used as a VBO by a subsequent vertex shader of the geometry processing stage. This approach increases integration flexibility by decoupling the morphing computation from the actual rendering. However, doing so requires additional data storage (due to padding and the additional result buffers) and introduces further state changes due to the compute pass. Algo. 4 shows the pseudocode implementation for the corresponding hook methods.

Algorithm 3 SSBO-based morphing implementation.

1: **procedure** COMPUTEMAPPING(L)
2:     **for all** $i \in [0, |L| - 2]$ **do**
3:         Buffer$_{M_i}$ ← MAPPCS($L[i], L[i+1]$)
4:         TRANSFERTOGPU(Buffer$_{M_i}$)
5:     **end for**
6: **end procedure**
7: **procedure** PERFORMMORPHING(t)
8:     RENDERPASS(t)
9: **end procedure**
10: **procedure** INITPCS(L)
11:     **for all** $S \in L$ **do**
12:         Buffer$_S$ ← CONSTRUCTBUFFER(S)
13:         PADBUFFER(Buffer$_S$)
14:         TRANSFERTOGPU(Buffer$_S$)
15:     **end for**
16: **end procedure**

Algorithm 4 CS-based morphing implementation.

1: **procedure** COMPUTEMAPPING(L)
2:     **for all** $i \in [0, |L| - 2]$ **do**
3:         Buffer$_{M_i}$ ← MAPPCS($L[i], L[i+1]$)
4:         TRANSFERTOGPU(Buffer$_{M_i}$)
5:     **end for**
6: **end procedure**
7: **procedure** INITMORPHSTEP(S, D)
8:     Buffer$_R$ ← INITBUFFEROFSIZE(max($|S|, |D|$))
9:     TRANSFERTOGPU(Buffer$_R$)
10: **end procedure**
11: **procedure** PERFORMMORPHING(t)
12:     COMPUTEPASS(t)
13:     RENDERPASS()
14: **end procedure**
15: **procedure** INITPCS(L)
16:     **for all** $S \in L$ **do**
17:         Buffer$_S$ ← CONSTRUCTBUFFER(S)
18:         PADBUFFER(Buffer$_S$)
19:         TRANSFERTOGPU(Buffer$_S$)
20:     **end for**
21: **end procedure**

7. Evaluation and Discussion

7.1. Performance Evaluation

Fig. 4 shows a base-line measurement of the average frametime when rendering a single PC without animation. The measured PCs resembles a sphere. Two different resolutions, one in windowed mode, one in full-screen mode, were tested. The measurements show that our system is able to render PCs with millions of points. Even for 100 million points, the frametime is with 50 ms perframe in full-screen mode sufficient for interaction (with approx. 20 FPS). Schütz *et al.* showed that the performance of rendering a PC can be increased significantly when using a compute-based rendering approach instead of the standard rasterization pipeline with `GL_POINT` primitives [SKW21].
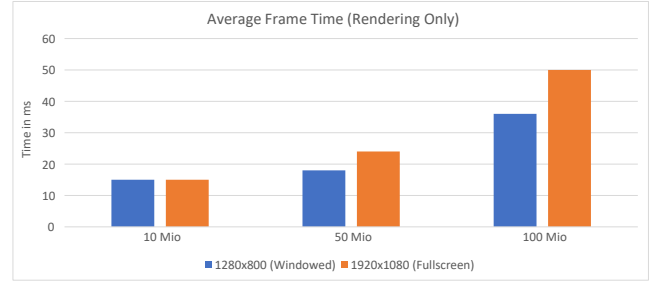

Figure 4: *Base-line measurements: average frametime (milliseconds) for rendering PCs of different sizes without animation.*

In our work, we nevertheless relied on the standard rasterization pipeline, as it allows for controlling the point size easily. Especially for the standard rasterization pipeline, Schütz *et al.* further propose to sort the points using shuffled Morton order to increase performance. However, this approach relies on spatial locality of points, which would be invalidated during animation and is therefore not applicable to our use-case. Note that the focus of Schütz *et al.* is on fast rendering of huge PCs, while we focus on animation. Our work is therefore complementary to theirs. Their compute-based rendering approach could be probably adapted and included into our framework for even faster rendering. We evaluate the performance of the different implementation approaches for PC morphing, described in Sec. 6.

Test Data & Test Setup. For testing purposes, we generate two PCs, resembling a unit cube and a unit sphere. For each of the PCs, three variants are generated with $|A| = 10^6$, $|B| = 10^7$, and $|C| = 2 \cdot 10^7$ points. Further, we use a common per-vertex configuration comprising position, color, and normal vector. The performance is measured for the following morphing animations: $A \xrightarrow{\mu} A$ (T.1), $B \xrightarrow{\mu} A$ (T.2), $B \xrightarrow{\mu} B$ (T.3), $C \xrightarrow{\mu} A$ (T.4), $C \xrightarrow{\mu} B$ (T.5), $C \xrightarrow{\mu} C$ (T.6). We only measure the performance of morphing the larger PC to the smaller PC. The morphing direction has no impact on the performance, as internally always the larger PC is used for rendering and the mapping computation is independent of the morphing direction. For the measurements, the camera is fixed in a way that the PCs are screen-filling and the viewing direction is not changed, as this can influence the performance (as observed by Schütz *et al.* [SKW21]). The output window had a resolution of 1280×800 pixels and rendering was performed without multi-sampling enabled.

The following timings were obtained: (1) *initialization time* denotes the time required to prepare all necessary buffers after a new PC has been loaded; (2) *mapping computation time* denotes the time to compute the point mapping for the different mapping approaches; (3) *average frametime* denotes the time to render frames during the morphing process. Each measurement was performed three times for each morphing sequence and the results were averaged (3 s window). The measurements were performed on an Intel Core i5-4460 processor (4 cores, 3.2 GHz), 16 GB RAM and a NVIDIA GeForce RTX 2080 Ti GPU (16 GB VRAM).

Table 1: *Comparison between the different implementation approaches for morphing animations.*

Implementation Aspect	VBO	SSBO	CS
Initialization time	low	medium	medium
Mapping computation time	high	medium	medium
Average frametime	stable	stable	unstable
Memory update/transfer	high	low	low
VRAM consumption	low	medium	high
Modularity	low	low	high

Test Results. Fig. 5(a) shows the initialization time for the different PC sizes and implementation approaches. Fig. 5(b) shows the mapping computation time, which includes the client-side computation of the mapping and the transfer of the result buffers to VRAM. The distance-based mapping approach was not included in the measurements as it is very compute intensive and not suitable to render morphing sequences in real-time. Fig. 5(c) shows the average frametime for the different mapping types and implementation approaches. Due to system characteristics, the minimum frametime is 15 ms. In addition to the performance measurements, Fig. 5(d) shows the required amount of memory that is transferred to the GPU on initialization of the PCs, on mapping computation, and the overall VRAM memory consumption. Using the VBO-based approach, each point requires 36 B for representation in VRAM. All points have to be transferred after mapping computation. Additionally, the size of the buffer for each PC is determined by the size of the larger PC, as the VBOs are required to have the same size. Using one of the other two approaches, each point requires 48 B as additional padding is required for correct buffer alignment, but the points have to be transferred to VRAM only once. During mapping computation, only the mapping data is transferred. For the compute shader approach, an additional result buffer is required, therefore the overall memory consumption is higher than the SSBO-based approach.

7.2. Discussion and Comparison of Approaches

Tab. 1 gives an overview of the three implementation approaches and their characteristics. For the VBO-based approach, no data is transferred to VRAM initially (Fig. 5(d)) and therefore no initialization time is measured (Fig. 5(a)). On mapping computation though, all points have to be transferred to VRAM, resulting in the higher processing time, compared to the other two approaches (Fig. 5(b)). The average frametime is stable, only for PCs with 20 million points, a small increase could be observed. Thus overall, the VBO-based approach is capable of providing interactive frame rates during the morphing itself, but has high mapping computation costs and memory transfer rates before the morphing can be performed. The SSBO-based approach improves on this and is able to reduce the mapping computation costs significantly while maintaining interactive frame rates. The initialization time (Fig. 5(a)) increases with the PC size, as all points are transferred to VRAM once during preprocessing. On mapping recomputation, only a small buffer containing the mapping data has to be transferred (Fig. 5(d)), which results in faster mapping-computation times (Fig. 5(b)). The overall memory consumption
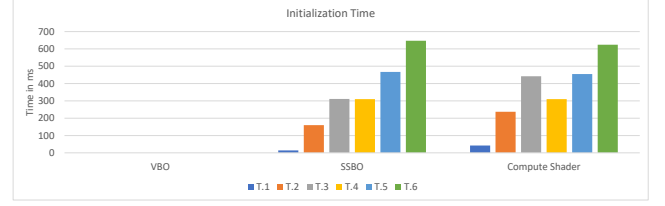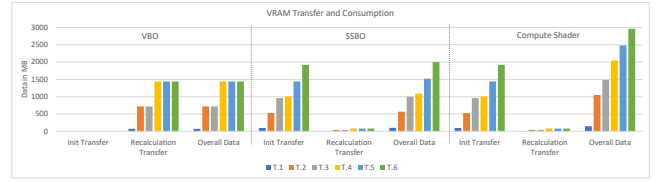

(a) Initialization time for different PCs sizes
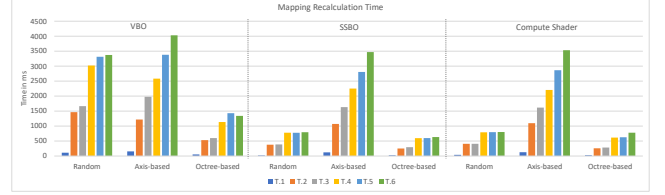
(b) Time required computing different mapping types
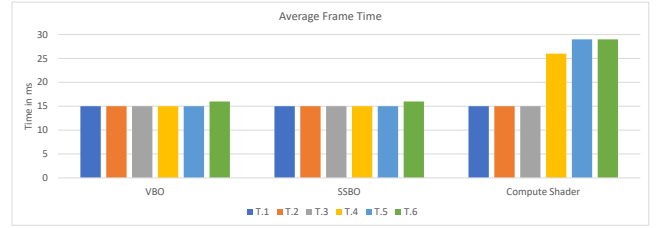
(c) Average frametime regarding PC sizes and mapping types

(d) Data transfer and VRAM consumption

Figure 5: *Measurement results for the different morphing approaches and morphing animations are shown (a)-(c), while (d) compares data transfer and VRAM consumption of the different morphing implementations and test animations.*

may be slightly higher compared to the VBO-based approach, due to padding. The implementation using compute shaders is similar to the SSBO-based approach, with respect to initialization and mapping recomputation costs. The overall memory consumption is higher (Fig. 5(d)), as an additional result buffer has to be stored. Using compute shaders, the frametime increases for larger PCs. There are probably two reasons for this: (1) the compute pass induces additional state changes. (2) the compute shader uses multiple SSBOs, which do not provide access as fast as a VBO. Nevertheless, this approach has the advantage of high modularity, due to the decoupling of morphing and rendering. This results in better reusability and maintainability.

To summarize, the SSBO-based implementation is well suited for rendering at interactive frame rates even when rendering large

PCs. Increased modularity can be achieved by means of CS, but interactive frame rates can only be provided for PCs smaller than 10 million points.

8. Conclusions & Future Work

This paper presents a GPU-aligned framework for rendering procedural animations of PCs. The decoupling of animation and rendering techniques enables simultaneous combinations of different animations with real-time rendering techniques for PCs. We present different application examples, such as attribute animations, explosion views, and PC morphing. The latter is explored in-depth, and possible implementation approaches are described and evaluated with respect to performance, memory usage, and modularity. The results show that an implementation based on compute shader provides high modularity, but is only feasible for smaller PCs. For larger PCs, using a vertex shader in combination with a SSBO is recommended.

This work can serve as a basis for advancements in several directions, e.g., to develop and prototype new animation techniques as well as for conducting further experiments and performance optimizations. Regarding this, the modularity offered by the compute shader approach is also desirable for huge PCs. However, for huge PCs the required frametime has to be reduced, to provide interactive framerates. Here, different configurations (e.g., of global and local workgroup sizes) could be evaluated. Additionally, compute-based rendering, as proposed by Schütz *et al.* could be adapted and evaluated for feasibility. Further, additional user interaction techniques could be explored in greater depth. Lastly and most important, the constraints of our system should be investigated further. In literature, cluster-based approaches are often used for computing point-to-point mappings. In the future, the system could be extended to enable cluster-based mapping approaches as well. Also, the maximum PC size that can be loaded and used within the system is currently limited by main memory.

References

[Ada06] ADAMS B.: *Point-Based Modeling, Animation and Rendering of Dynamic Objects*. PhD thesis, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium, May 2006. 2

[ARLP18] ANJOS R. K., RIBEIRO C. S., LOPES D. S. O., PEREIRA J. A. M.: Stroke-based splatting: An efficient multi-resolution point cloud visualization technique. *Vis. Comput. 34*, 10 (Oct. 2018), 1383–1397. 2

[BGQ05] BAO Y., GUO X., QIN H.: Physically based morphing of point-sampled surfaces. *Comput. Animat. Virtual Worlds 16*, 3-4 (2005), 509–518. 2

[CHG*20] CHEN Y., HU V. T., GAVVES E., MENSINK T., METTES P., YANG P., SNOEK C. G. M.: Pointmixup: Augmentation for point clouds, 2020. 2

[CSY*20] CHENG Q., SUN P., YANG C., YANG Y., LIU P.: A morphing-based 3d point cloud reconstruction framework for medical image processing. *Computer Methods and Programs in Biomedicine 193* (04 2020), 105495. 3

[CU03] CMOLIK L., ULLER M.: Point cloud morphing. In *7th Central European Seminar on Computer Graphics (CESCG)* (Apr. 2003). 2, 4

[DJKM17] DHARMA D., JONATHAN C., KISTIJANTORO A. I., MANAF A.: Material point method based fluid simulation on gpu using compute shader. In *2017 International Conference on Advanced Informatics, Concepts, Theory, and Applications (ICAICTA)* (08 2017). 2

[Ebe14] EBERT D. S.: *Texturing and Modeling: A Procedural Approach*. Academic Press, Inc., USA, 2014. 1

[Har62] HARLOW F. H.: The particle-in-cell method for numerical solution of problems in fluid dynamics. 2

[IKS20] IZDEBSKI Ł., KOPIECKI R., SAWICKI D.: Bézier curve as a generalization of the easing function in computer animation. In *Advances in Computer Graphics* (2020), Magnenat-Thalmann N., Stephanidis C., Wu E., Thalmann D., Sheng B., Kim J., Papagiannakis G., Gavrilova M., (Eds.), Springer International Publishing, pp. 382–393. 4

[MKN*04] MÜLLER M., KEISER R., NEALEN A., PAULY M., GROSS M., ALEXA M.: Point based animation of elastic, plastic and melting objects. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Goslar, DEU, 2004), SCA '04, Eurographics Association, p. 141–151. 2

[OFCD02] OSADA R., FUNKHOUSER T., CHAZELLE B., DOBKIN D.: Shape distributions. *ACM Trans. Graph. 21*, 4 (Oct. 2002), 807–832. 3

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., p. 335–342. 2

[SKW21] SCHÜTZ M., KERBL B., WIMMER M.: Rendering point clouds with compute shaders and vertex order optimization. *CoRR abs/2104.07526* (2021). 4, 6

[SPB*19] SABBADIN M., PALMA G., BANTERLE F., BOUBEKEUR T., CIGNONI P.: High dynamic range point clouds for real-time relighting. *Computer Graphics Forum 38*, 7 (2019), 513–525. 1

[SW15] SCHÜTZ M., WIMMER M.: High-Quality Point Based Rendering Using Fast Single Pass Interpolation. In *International Congress on Digital Heritage - Theme 2 - Computer Graphics And Interaction* (2015), Guidi G., Scopigno R., Brunet P., (Eds.), IEEE. 1

[THCF06] TIAN H., HE Y., CAI H., FENG L.: Efficient metamorphosis of point-sampled geometry. In *16th International Conference on Artificial Reality and Telexistence–Workshops (ICAT'06)* (2006), pp. 260–263. 2

[TZZ09] TAN G., ZHANG S., ZHANG Y.: Shape morphing for point set surface based on vertex deformation gradient. In *2009 WRI World Congress on Software Engineering* (2009), vol. 2, pp. 466–471. 2

[WZH12] WANG R., ZHANG C., HU J.: Smooth morphing of point-sampled geometry. In *Computer Applications for Graphics, Grid Computing, and Industrial Environment - International Conferences, GDC, IESH and CGAG* (2012), Kim T., Cho H. S., Gervasi O., Yau S. S., (Eds.), vol. 351 of *Communications in Computer and Information Science*, Springer, pp. 16–23. 2

[XLF*19] XI R., LUO Z., FENG D. D. F., ZHANG Y., ZHANG X., HAN T.: Survey on smoothed particle hydrodynamics and the particle systems. *IEEE Access PP* (12 2019), 1–1. 2

[XZPF04] XIAO C., ZHENG W., PENG Q., FORREST A. R.: Robust morphing of point-sampled geometry. *Comput. Animat. Virtual Worlds 15*, 3-4 (2004), 201–210. 2